UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Miha Herblan

Zagotavljanje kakovosti programskih rešitev po naročilu

DIPLOMSKO DELO
NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

Mentor: pred. mag. Igor Škraba

Ljubljana, 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V nalogi predstavite posebnosti programskih rešitev po naročilu ter standarde in postopke za zagotavljanje kakovosti programske opreme. Opišite načela in metode pri testiranju programske opreme, ki je eden od korakov pri zagotavljanju kakovosti. Opišite praktičen primer programske rešitve po naročilu, izdelajte testni scenarij in analizirajte rezulteta testiranja.

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Miha Herblan,

z vpisno številko 63050252,

sem avtor diplomskega dela z naslovom:

Zagotavljanje kakovosti programskih rešitev po naročilu.

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom pred. mag. Igor Škraba,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov.,angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 29. 5. 2014 Podpis avtorja:

Zahvala

Zahvaljujem se mentorju, pred. mag. Igorju Škrabi, za vso pomoč pri izdelavi diplomske naloge.

Zahvaljujem se svoji družini za podporo v času študija.

Kazalo

Povzetek

V diplomski nalogi je obravnavana tematika zagotavljanja kakovosti programskih rešitev po naročilu. Najprej si bomo pogledali življenjski cikel razvoja programske opreme. Nato bodo podrobneje predstavljeni zagotavljanje kakovosti, standardi, ki stojijo za tem, kaj sploh je zagotavljanje kakovosti programske opreme, kako kakovost merimo in predvsem zakaj izvajamo zagotavljanje kakovosti. Za tem so predstavljena načela, ki veljajo na splošno pri zagotavljanju kakovosti. Glede na to, da je zagotavljanje kakovosti precej širok pojem, si bomo natančneje pogledali testiranje programske opreme. Ker se osredotočamo na programske rešitve po naročilu, so predstavljene posebnosti tega področja. Sledil bo pregled najbolj uveljavljenih načinov zagotavljanja kakovosti. Vse skupaj pa je zaključeno s praktičnim primerom programske rešitve po naročilu, ki zavzema pripravo uporabniških zahtev, zasnovo rešitve in testnega scenarija.

Ključne besede:

zagotavljanje kakovosti, testiranje programske opreme, metode testiranja, programske rešitve po naročilu

Abstract

In thesis we look at problem of software quality assurance, especially when it comes to custom solutions projects. First we look at what quality assurance is, how do we mesure it and specialty why we do it. Afterwards we go through main principles, that apply when dealing with quality assurance in general. Since quality assurance is extensive topic, we take more detailed look on one part of quality assurance that is mostly used, that is testing of software. Because we are talking about software custom solution projects, we need to familiarize yourself with process that goes with that. Then we go through some of the most common practises of testing. In final chapter we take actual custom solution project and we take through the whole proces.

Key words:

Quality assurance, software testing, testing methods, software custom solution projects

Poglavje 1

Uvod

V zadnjih nekaj desetletjih so računalniški sistemi postali pomemben del našega vsakdanjika. Skupaj z računalniškimi sistemi se je začela pospešeno razvijati tudi pripadajoča programska oprema. Z na eni strani vedno večjim povpraševanjem in posledično željo po čim krajših dobavnih rokih, na drugi strani pa z vse večjim vplivom na naše življenje, se je izkazala potreba po zagotovitvi določene kakovosti programske opreme.

Posledice slabega zagotavljanja kakovosti ali celo njena odsotnost so lahko vzrok tragičnim dogodkom. Poglejmo si nekaj primerov[1]:

- Napaka v programski opremi naprave za obsevanje je povzročila smrt nekaterih pacientov v osemdesetih letih.
- Leta 1996 je napaka na računalniški krmilni enoti povzročila uničenje rakete Ariane 5 manj kot minuto po izstrelitvi.
- 14. avgusta 2003 je napaka v programski opremi za nadzor omrežja povzročila izpad električne energije na širšem območju severovzhodnega dela Severne Amerike.
- Spletni iskalnik Google je zaradi napake januarja 2009 vse spletne strani označil kot sumljive, vključno s svojo.

Research Triangle Institute (RTI) je za urad National Institute of Standards and Technology (NIST) leta 2002 v raziskavi ugotovil, da napake v programski opremi stanejo ameriško gospodarstvo skoraj 60 milijard ameriških dolarjev letno oziroma 0.6 odstotka bruto domačega proizvoda[2].

Kljub vse večji in vse bolj pestri ponudbi programske opreme se vedno več kupcev te opreme odloča za dodelave, spremembe in prilagoditve po meri. Ker

zaradi posameznih in običajno različnih želja različnih kupcev ne gre spreminjati produkta, se te dodatne želje naredijo v procesu rešitve po naročilu. Seveda je treba, tako kot za produkt, tudi za rešitev po naročilu zagotoviti določeno stopnjo kakovosti. Rešitev po naročilu mora zadovoljiti želje naročnika, obenem pa mora ustrezati internim smernicam razvoja in zagotavljanja kakovosti. Hkrati je treba paziti, da se že dosežena kakovost produkta ne poslabša.

Vsaka rešitev po naročilu se začne z zajemom uporabniških zahtev, oceno vrednosti projekta in oceno želenega datuma dobave rešitve. Če so ti podatki sprejemljivi, se lahko začne projekt razvoja rešitve po naročilu. Pripravi se dokument zasnove rešitve, ki natančno opisuje delovanje rešitve s tehničnega in uporabniškega vidika. Osnova za pripravo zasnove rešitve je dokument uporabniških zahtev. Zasnovo rešitve je treba uskladiti z naročnikom, ki jo na koncu tudi potrdi. Na podlagi zasnove rešitve se pripravi ponudba za naročnika. Po potrditvi ponudbe se začne z delom.

Poleg programiranja se čim prej pripravijo tudi testni scenariji. Testni scenariji se pripravijo na podlagi dokumentov uporabniških zahtev in zasnove rešitve. Pomemben faktor pri pripravi kakovostnega testnega scenarija so tudi izkušnje osebe, ki ga pripravlja. Če je mogoče, se testiranje začne takoj, ko je pripravljen vsaj del rešitve. To predstavlja smiselno zaključeno celoto. Po zaključku programiranja se izvede test za celotno rešitev po pripravljenem testnem scenariju. Če se med testiranjem pojavijo napake, se način ponovitve napake zabeleži in sporoči osebi, ki je ta del programamirala. Ko rešitev prestane vse testne scenarije brez težav, se dobavi stranki.

Poglavje 2

Zagotavljanje kakovosti programske opreme

Če želimo razumeti namen in razloge za testiranje programske opreme, si moramo najprej pogledati, kako razvoj programske opreme sploh poteka.

2.1 Življenjski cikel razvoja programske opreme

Življenjski cikel razvoja programske opreme se začne z zasnovo in konča z vzdrževanjem pri uporabniku. Razvojni proces je razčlenjen na zaporedje medsebojno odvisnih aktivnosti, ki temeljijo na začetnih potrebah po izdelavi uporabnega produkta. Zakaj se govori o "ciklu"? Vsak razviti produkt ustvarja nove potrebe, ki zahtevajo razvoj novih izdelkov[3].

Definicija življenjskega cikla:

Življenjski cikel programske opreme je nabor diskretnih aktivnosti v času razvoja programske opreme in programskih sistemov. Faza življenjskega cikla je čas izvajanja posameznih aktivnosti (tudi sama aktivnost)[3].

Življenjski cikel programske opreme je tipično razdeljen na naslednje faze[4]:

Faza 1 Analiza zahtev in specifikacija sistema.

Ta faza vključuje analiziranje programskega problema (funkcionalni opis) in specifikacije želenega obnašanja sistema (funkcionalne zahteve in specifikacije). Rezultat je dokument z imenom: Specifikacije zahtev programske opreme (SZPO oz. ang. Software Requirements Specifications).

Aktivnosti te faze analize ločimo v 2 skupini:

- analiza problema - rezultat je popolno razumevanje problemskega področja,
- opis produkta - rezultat je skladen in celovit dokument programskih specifikacij (SZPO).

Aktivnosti obeh skupin ne izvajamo zaporedno, temveč sočasno.

SZPO opisuje funkcionalne zahteve, značilnosti strojnega okolja, osnovno obliko uporabniških vmesnikov in performančne cilje oz. zahtevane zmogljivosti.

V tej fazi razvijalci in uporabniki dobijo odgovor na vprašanje: "Kaj potrebujemo oz. kaj naj bi grajeni programski sistem zagotavljal?"

Faza 2 Načrtovanje sistema in komponent.

Vključuje preliminarno načrtovanje - izvedeta se razčlenitev programskega sistema na njegove dejanske konsistentne komponente in interaktivna razgradnja teh komponent v vedno manjše podkomponente, dokler niso dovolj majhne, da jih lahko ljudje brez težav razumejo. Vsak modul je dokumentiran - opisani so vhodi, izhodi in funkcije preoblikovanja podatkov. Za vsak modul definiramo in dokumentiramo algoritme.

Kot rezultat te faze so narejeni:

- modularna razgradnja programskega sistema,
- definicija strukture podatkov,
- definicija formata datotek in
- opis pomembnejših algoritmov.

V tej fazi odgovorimo na vprašanje: "Kako narediti oz. kako bomo zadovoljili identificirane zahteve?"

Faza 3 Implementacija in testiranje komponent sistema.

Izvede se kodiranje algoritmov. Algoritmi se transformirajo v računalniku razumljiv jezik (praviloma se uporabljajo višji programski jeziki). Testirajo in odpravljajo se napake vsakega modula, specificiranega pri načrtovanju.

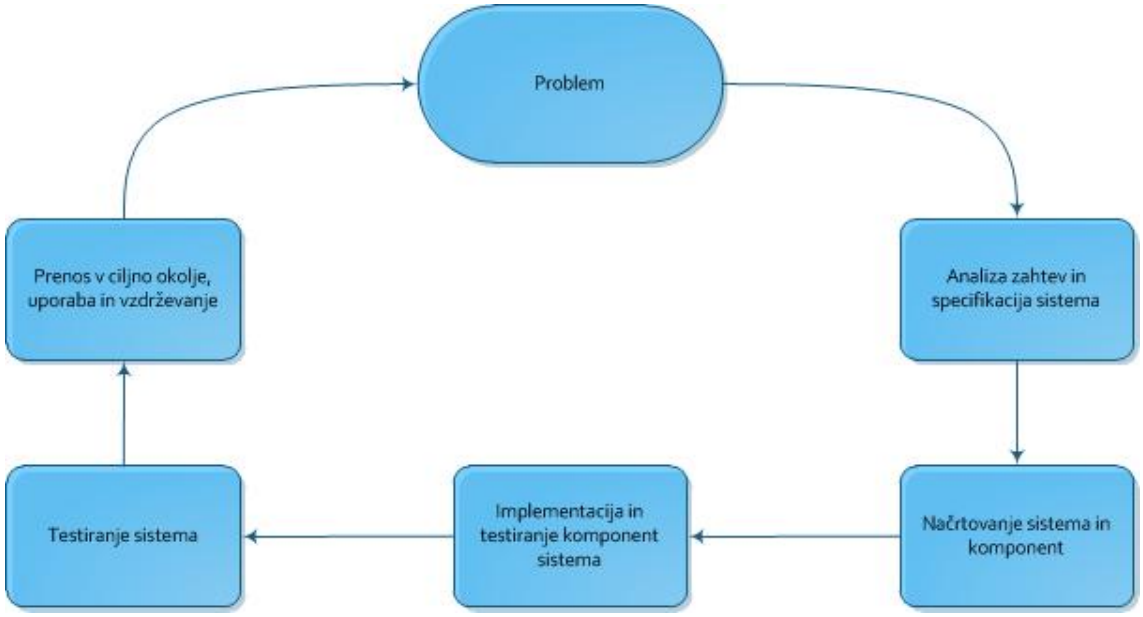
Faza 4 Testiranje sistema.

Začnemo s testiranjem posameznih modulov, tako se osredotočimo na del programa, da lažje ugotovimo in odstranimo napake. Hkrati kontroliramo tudi obnašanje modula glede na podane specifikacije (funkcionalno testiranje).

Po tej fazi že preverjene module integriramo oz. povežemo v enotno programsko strukturo ter jih testiramo kot celoto. Sledi še sistemsko testiranje, s katerim preverimo, ali se celoten programski sistem, postavljen v določeno strojno okolje, obnaša ustrezno podanim specifikacijam zahtev programske opreme.

Faza 5 Prenos v ciljno okolje, uporaba in vzdrževanje.

Programska oprema in pripadajoča dokumentacija se izročita uporabnikom. Začne se uporaba sistema. V primeru napak ali pomanjkljivosti se pripravijo in implementirajo ustrezni popravki ter nadgradnje sistema.



Slika 2.1: Življenjski cikel razvoja programske opreme.

2.2 Zagotavljanje kakovosti

Izraz zagotavljanje kakovosti je uporabljen na več področjih in v več industrijah ter panogah po vsem svetu. V splošnem bi lahko izraz predstavili kot sistematične in planirane aktivnosti, ki jih izvedemo ob izdelavi izdelka ali realizaciji storitve, da preverimo skladnost z zahtevami, ki so bile podane[5].

S standardiziranjem zagotavljanja kakovosti se ukvarja več standardov, ki so združeni v sklop mednarodnih standardov SQuaRE (ang. Systems and

software Quality Requirements and Evaluation). Celoten sklop sestavljajo naslednje družine standardov:

- družina standardov za upravljanje kakovosti (ISO/IEC 2500n),
- družina standardov za model kakovosti (ISO/IEC 2501n),
- družina standardov za merjenje kakovsti (ISO/IEC 2502n),
- družina standardov za zahteve kakovosti (ISO/IEC 2503n) in
- družina standardov za ocenjevanje kakovosti (ISO/IEC 2504n).

Trenutni aktualni standard za model zagotavljanje kakovosti je ISO/IEC 25010:2011. Ta standard nadomešča predhodni standard ISO/IEC 9126-1:2001.

Model kakovosti produktov je sestavljen iz osmih značilnosti (funkcionalna ustreznost, zanesljivost, učinkovitost, zmogljivost, uporabnost, varnost, združljivost, vzdrževanje in prenosljivost)[6].

1. Funkcionalna ustreznost

   Nivo, s katerim produkt ali sistem opravlja funkcije, ki so predvidene in navedene v zahtevah.

2. Učinkovitost izvajanja

   Zmogljivost glede na količino sredstev, ki se uporabljajo v skladu z navedenimi pogoji.

3. Uporabnost

   Nanaša se na nivo potrebnega napora, da se uporabniki naučijo uporabe, priprave vhodnih podatkov in razumevanja izhodnih podatkov programske opreme.

4. Združljivost

   Nivo, v kolikšni meri produkt, sistem ali njegov del lahko izmenjuje podatke z drugimi produkti, sistemi ali komponentami in/ali opravlja predvidene naloge, medtem ko si delijo isto strojno in/ali programsko okolje.

5. Zanesljivost

   Nivo, na katerem se pričakuje, da bo programska oprema opravljala svoje zahtevane naloge na podlagi navedenih pogojev za določeno obdobje.

6. Varnost

   Nivo, s katerim produkt ali sistem ščiti informacije in podatke, tako da osebe, funkcije in sistemi lahko dostopajo samo do podatkov, za katere imajo dovoljenje.

7. Vzdrževanje

   Nivo uspešnosti in učinkovitosti, s katerim sistem, produkt ali funkcijo spreminjamo glede na predvidena vzdrževanja.

8. Prenosljivost

   Nivo uspešnosti in učinkovitosti, s katerim sistem, produkt ali funkcijo prestavimo z ene strojne opreme in/ali programske opreme na drugo.

Zagotavljanje kakovosti se izvaja v vseh fazah življenskega cikla razvoja programske opreme.

2.3 Merjenje kakovosti

O tem, kaj je kakovost, sta zapisani dve trditvi v IEEE Standard Glossary of Software Engineering Terminology[12].

1. Kakovost je povezana s stopnjo, s katero sistem, del sistema ali proces ustreza specificiranim zahtevam.

2. Kakovost je povezana s stopnjo, s katero sistem, del sistema ali proces ustreza strankinim ali uporabniškim potrebam ali željam.

Da določimo, ali je sistem, del sistema ali proces visoke kakovosti, uporabljamo lastnosti, ki so opisane v poglavju 2.2. Te značilnosti programske opreme so funkcionalna ustreznost, učinkovitost izvajanja, uporabnost, združljivost, varnost, vzdrževanje in prenosljivost.

K tem značilnostim pa lahko dodamo še eno značilnost programske opreme, tj. zmožnost izvedbe dobrega testa (testabilnost). Ta atribut zadeva bolj razvojnika/testerja kot naročnika/stranko. Pri ugotavljanju testabilnosti programske opreme si pomagamo z dvema vprašanjema[11]:

1. Kolikšen napor je potreben za testiranje programske opreme, da potrdimo delovanje glede na specifikacijo?

2. Kakšno zmožnost ima programska oprema, da razkrije pomanjkljivosti med izvajanjem testiranja?

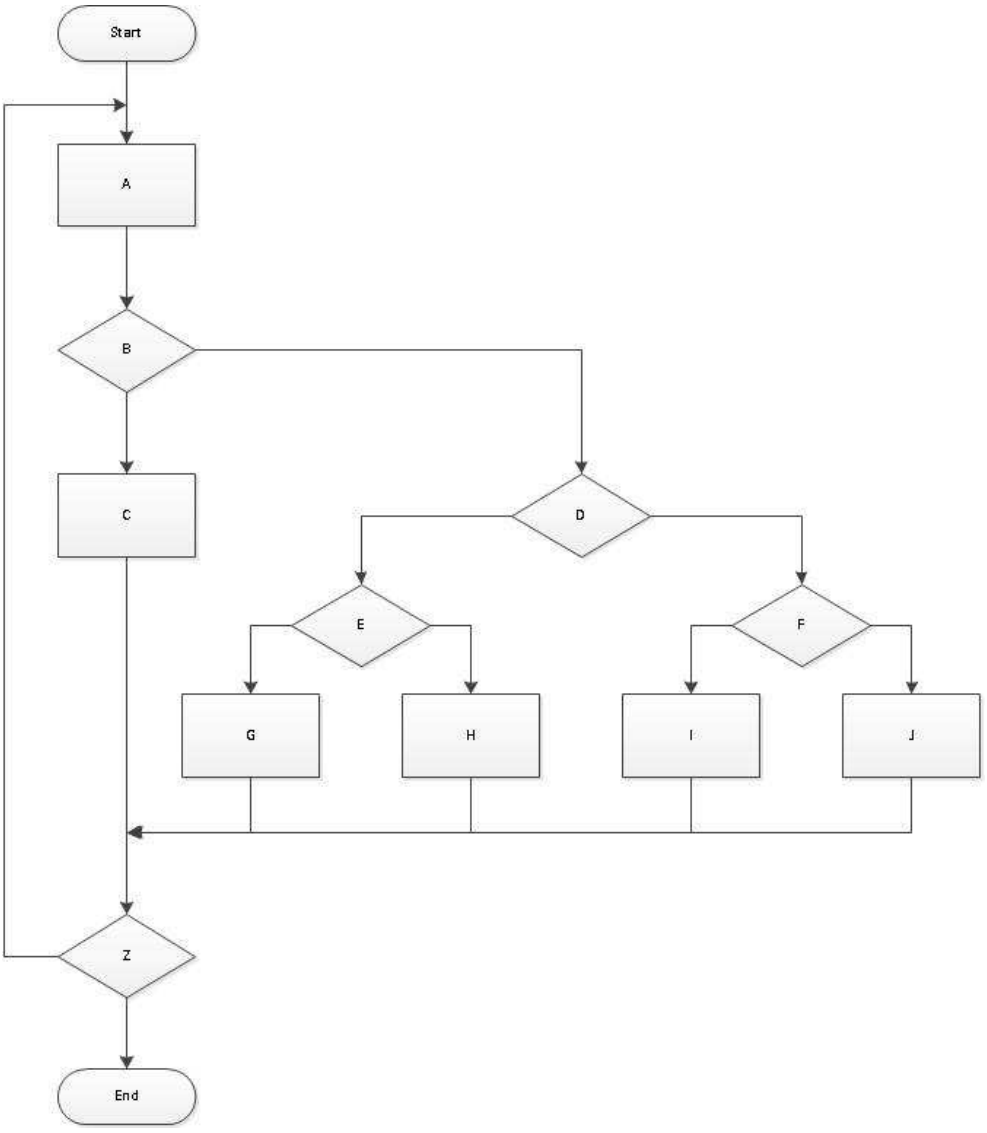
2.4 Zakaj ni mogoče zagotoviti stoodstotne pravilnosti delovanja

Da bi lahko z gotovostjo trdili, da je programska oprema brez napake, bi morali preveriti vse mogoče kombinacije izvajanja programa, kar pa je nemogoče. Zakaj je tako, si poglejmo na naslednjem primeru[14].

Na sliki 2.2 je predstavljen diagram poteka enostavnega programa. Od začetka (Start) do konca (End) programa je možnih 5 različnih poti.

- "ABCZ"
- "ABDEGZ"
- "ABDEHZ"
- "ABDFIZ"
- "ABDFJZ"

Vzemimo primer, da se pri pogoju Z na začetek lahko vrnemo 20-krat. Ko se skozi zanko sprehodimo prvič, imamo 5 možnih poti. Pri dveh ponovitvah jih je 5^2, kar predstavlja 25 možnih poti. Če imamo 3 ponovitve, število naraste na 125 (5^3). Kot lahko vidimo, število možnih poti s številom ponovitev eksponentno narašča. Tako imamo po dvajsetih ponovitvah ogromno število možnih kombinacij, ki jih ni mogoče preveriti v razumnem času.



Slika 2.2: Diagram poteka enostavnega programa.

Poglavje 3

Testiranje programske opreme

3.1 Definicija testiranja

"Testiranje programske opreme je proces izvajanja programa z namenom iskanja napak"[10].
Eden glavnih vzrokov za slabo testiranje je napačna predpostavka, s katero začne večina razvijalcev, ki navadno rečejo:

- "Testiranje je dokazovanje, da napake niso prisotne."
- "Cilj testiranja je dokazati, da program deluje, kot je bilo predvideno."
- "Testiranje je proces vzpostavitve zaupanja, da program deluje, kot mora."

Z upoštevanjem zgornjih predpostavk pri testiranju je manjša verjetnost odkrivanja napak v programski opremi, zato bomo izbirali vhodne podatke na osnovi prepričanja, da napake niso prisotne. To pa ne velja za praktično vsak program. Zato je treba za cilj testiranja postaviti prvotno trditev, ki povzroči izbiranje takšnih vhodnih podatkov, ki bodo z večjo verjetnostjo povzročili nepravilno delovanje programa[10].

3.2 Načela testiranja

Pri testiranju programske opreme se lahko naslonimo na pretekle izkušnje in primere dobre prakse. Spodnjih enajst točk povzema načela, ki se jih je treba držati, če želimo doseči visok nivo kakovosti.[11]

Načelo 1

Testiranje je proces izvajanja programske opreme na podlagi izbranih testnih primerov z namenom (i) razkritja napak in (ii) ocenjevanja kakovosti.

Razvojniki so naredili velik napredek pri metodah, ki odpravljajo napake pri razvoju programske opreme. Kljub temu se napake pojavljajo in imajo negativen učinek na kakovost programske opreme. Osebe, ki izvajajo test, morajo te napake odkriti pred izdajo programske opreme. To načelo se zavzema za izvajanje programske opreme z namenom odkrivanja napak. Prav tako se zavzema za ločitev procesa testiranja od procesa razhroščevanja, saj je namen slednjega najti napake in jih tudi odpraviti. Termin "komponenta programske opreme", ki je uporabljen v kontekstu, lahko predstavlja kateri koli del programske opreme glede na velikost in kompleksnost, na primer od posameznih metod ali funkcij do celotnih sistemov. Termin "napaka", uporabljen v tem in naslednjih načelih, predstavlja kakršno koli odstopanje programske opreme, ki bi imelo negativne posledice na funkcionalnost, zmogljivost, zanesljivost, varnost in/ali katero od drugih definiranih meril kakovosti. Bertolino, v knjigi Guide to the Software Engineering Body of Knowledge, opiše testiranje kot "dinamični proces izvajanja programa z vrednotenjem vhodnih parametrov". To nam daje misliti, da pri testiranju ne gre zgolj za odkrivanje napak, ampak tudi za ocenjevanje kakovosti programske opreme.

Načelo 2

Pri testiranju z namenom odkrivanja napak je dober testni scenarij tisti, ki ima veliko verjetnost, da razkrije še neodkrite napake.

Drugo načelo se zavzema za natančno načrtovanje testnih scenarijev in podaja merilo, s katerim lahko ocenimo testne scenarije in učinkovitost porabe časa za testiranje, ko je cilj testiranja odkrivanje napak. To zahteva od osebe, ki testira, da premisli o ciljih vsakega testnega scenarija posebej, kar pomeni, da definira, katere napake predvideva, da jih bo odkril z določenim testnim scenarijem. Tukaj je pristop do dela osebe, ki izvaja test, podoben pristopu znanstvenika, ki izvaja poskus. Znanstvenik poskuša z izvajanjem poskusa potrditi ali ovreči postavljeno hipotezo. Glede na hipotezo so izbrani vhodni parametri in predpostavljeni pravilni izhodni podatki, nato se izvede test. Rezultati se primerjajo s

predpostavljenimi in na podlagi tega se potrdi oziroma ovrže hipotezo.

Načelo 3

Rezultate testov je treba skrbno pregledati.

Oseba, ki izvaja test, mora natančno pregledati in interpretirati testne rezultate. V nasprotnem primeru lahko pride do večjih in dražjih težav. Na primer:

- Napaka se spregleda in testni scenarij se označi kot opravljen. Testni proces se nadaljuje na napačni predpostavki, da predhodno opravljen testni scenarij ni razkril nobenih napak. Napaka se sicer lahko odkrije kasneje med izvajanjem drugih testov, vendar je popravilo take napake precej dražje, kot bi bilo, če bi bila napaka odkrita pravočasno.
- Pri pregledu testnih rezultatov lahko pride tudi do označitve napake, ki pa dejansko ne obstaja. V tem primeru se lahko porabi več napora za iskanje in odpravljanje neobstoječe napake. Šele ob ponovnem pregledu testnih rezultatov se odkrije, da napaka dejansko ne obstaja.
- Rezultat testa je lahko napačno interpretiran, kar ima za posledico ponavljanje nepotrebnega dela ali pa se spregleda kritična napaka.

Načelo 4

Testni scenarij mora vsebovati pričakovane rezultate izvajanja programske opreme.

Začetnik, ki izvaja test, tipično pričakuje v testnem scenariju vhodne podatke za izvedbo testa. Testni scenarij pa je brez pomena, če ne vsebuje tudi natančnih definicij rezultatov izvajanja testa. Na primer: rezultat izvajanja te funkcije s temi vhodnimi parametri je toliko ali pa ob pritisku na gumb se prikažejo naslednji podatki. Rezultat izvajanja testa za dane vhodne podatke osebi, ki izvaja test, omogočijo, da enostavno ugotovi, ali je test uspešno opravljen ali ne. Zelo pomembno je, da so rezultati izvajanja testa zapisani enolično in enostavno berljivo, da se izognemo izgubi časa zaradi napačne interpretacije rezultatov izvajanja testa. Testni scenarij mora torej vsebovati tako vhodne kot izhodne podatke.

Načelo 5

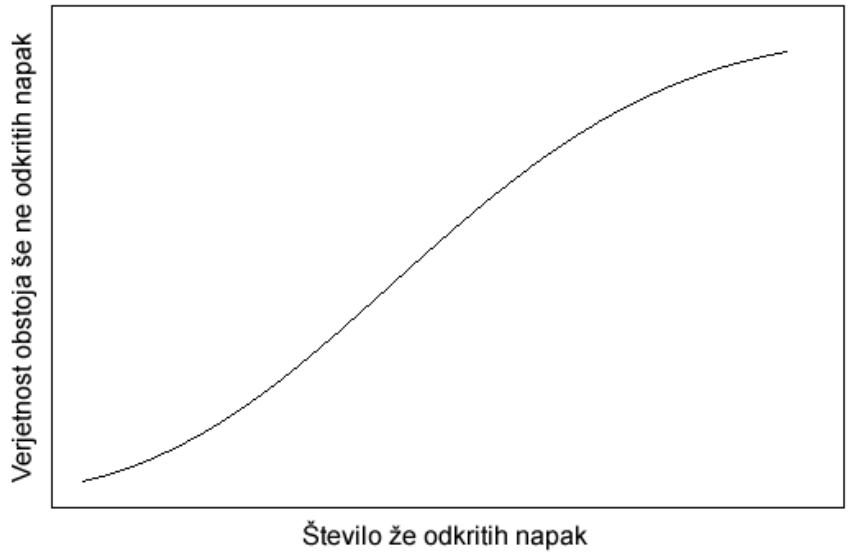
Testni primeri naj bi bili narejeni za veljavne in neveljavne primere.

Oseba, ki izvaja test, ne sme sklepati, da bodo imeli vsi testni primeri pravilne vhodne podatke. Napačni vhodni podatki se lahko pojavijo iz več razlogov. Uporabnik je na primer narobe razumel navodila ali pa ni seznanjen s pričakovanimi vhodnimi podatki v program. Prav tako se pogosto dogajajo napake pri vnosu podatkov, tudi če uporabnik popolnoma razume delovanje programa. Testni primeri, ki imajo za predvidene napačne vhodne podatke, so lahko zelo učinkoviti pri razkrivanju napak, ker povzročijo izvajanje programa, kot ni bilo predvideno, in to lahko pokaže nenavadno obnašanje programa. To načelo, tako kot tudi načelo 7, se tudi zavzema za neodvisne testne skupine iz naslednjega razloga: razvijalec programske opreme je lahko pristranski pri izbiri vhodnih podatkov v testnih primerih v taki meri, da program deluje pravilno. Neodvisen tester je bolj primeren pri izbiri vhodnih podatkov, ki so tudi napačni.

Načelo 6

Verjetnost odkritja dodatnih napak v programski opremi je proporcionalna številu že odkritih napak.

To pravilo pravi, da več kot je že odkritih napak v programski opremi, večja je verjetnost, da se odkrijejo še dodatne napake v nadaljnjih testih. Vzemimo za primer dva programa A in B. Testerji so odkrili 20 napak v programu A in 3 v programu B. Verjetnost, da obstajajo še neodkrite napake v programu A, je višja kot za program B. To empirično dognanje izhaja iz več razlogov. Napake se pogosteje pojavljajo v skupinah v bolj kompleksnem programu, ki je slabo načrtovan. V primeru takega programa se je treba odločiti, ali se izvede popolna prenova programa ali pa se poveča sredstva namenjena testiranju in se s tem zagotovi skladnost z zahtevami. To je še posebej pomembno pri varnostno kritičnih funkcijah.



Slika 3.1: Presenetljiva povezava med številom odkritih in neodkritih napak.

Načelo 7

Testiranje naj bi bilo izvedeno s strani neodvisnih skupin.

To načelo drži zaradi psiholoških in tudi zaradi praktičnih razlogov. Razvijalcu programske opreme je težko priznati, da ima program, ki ga je naredil, napako. Zato se mora oseba, ki izvaja test, zavedati, da imajo razvijalci svoj ponos in da je njim napako težko odkriti, ker mentalna predstava kode prevlada nad dejansko. Prav tako lahko razvijalec zahteve razume drugače. Tudi ko test odkrije napako, imajo razvijalci pogosto težavo z odkrivanjem napake, ker je njihov pogled na program zasenčen s kodo, ki jo imajo v mislih. Potrebo po samostojni in neodvisni skupini za zagotavljanje kakovosti lahko interpretiramo na več načinov. Eden od načinov je, da je testna skupina lahko povsem ločen oddelek v podjetju. Lahko je del skupine za zagotavljane kakovosti, lahko je tudi del razvojne ekipe. V tem primeru je treba biti pozoren, da je skupina neodvisna in objektivna. Člani te skupine naj bi se primarno posvečali testiranju in ne razvijanju novih rešitev. Za dobre rezultate je seveda pomembno dobro sodelovanje med testerji in razvijalci. Zato je potrebno

dobro medsebojno razumevanje.

Načelo 8
Testi morajo biti ponovljivi in ponovno uporabni.

Načelo 2 postavi vzporednice med delom osebe, ki izvaja test, in znanstvenika, ki izvaja poskus. Načelo 8 zahteva, da se vsa stanja, posebnosti, uporabljena oprema in rezultati zabeležijo. Ti podatki so namreč neprecenljivi za razvojnike, ko poskušajo ponoviti napako pri razhroščevanju. Prav tako so ti podatki pomembni pri ponovnem testiranju po odpravi napake. Ponovljivost in ponovna uporaba testov sta pomembni tudi pri testiranju novih izdaj programske opreme. Tako kot znanstveniki pričakujejo, da bodo drugi ponovili isti eksperiment na podlagi njihovih zapisov, mora oseba, ki izvaja test, pričakovati isto.

Načelo 9
Testiranje naj bi bilo načrtovano.

Testni plani naj bi bili pripravljeni za vsako fazo testiranja, cilji testiranja pa prav tako vnaprej pripravljeni in opisani obsežno, kolikor se le da. Testni plani, skupaj s cilji testiranja, so nujni, da se zagotovi potreben čas in da se alocira določene vire, potrebne za izvedbo testiranja. Testne plane je treba izvajati skozi cel življenjski cikel programske opreme. Načrtovanje testov mora biti usklajeno s projektnim planiranjem. Vodja testiranja mora sodelovati z vodjo projekta in skupaj morata usklajevati aktivnosti. Testerji ne morejo testirati dela ali celote programske opreme, če je razvijalci prej ne pripravijo. Nevarnosti pri testiranju morajo biti ocenjene. Kako verjetna je na primer zamuda dobavnega roka novega dela programske opreme, kateri deli programske opreme so kompleksnejši in zahtevajo dodatno testiranje, ali testerji potrebujejo dodatno izobraževanje za uporabo novih orodij? Pazljivo načrtovanje testov pomaga, da se izognemo nepotrebnim testom in neproduktivnemu ter neplaniranemu testiranju, popravljanju napak in ponovnem testiranju, kar velikokrat vodi do slabe programske opreme in težav pri pravočasni dobavi ter v okviru predvidenih stroškov.

Načelo 10
Testne aktivnosti naj bi bile del razvojnega cikla programske opreme.

Odlašanje z začetkom testiranja do konca razvoja ni primerno. Načrtovanje testnih aktivnosti mora biti del razvojnega cikla programske opreme že od začetka. Začne se z analizo uporabniških zahtev in se nadaljuje skozi celoten razvojni cikel vzporedno z razvojnimi aktivnostmi. Poleg načrtovanja testiranja se lahko izvedejo tudi ostali načini testiranja, kot je na primer test uporabnosti z uporabo prototipov. Te aktivnosti se lahko nadaljujejo, dokler ni programska oprema predana uporabnikom.

Načelo 11

Testiranje zahteva kreativnost in iznajdljivost.

Oseba, ki izvaja test, se srečuje z naslednjimi težavami in izzivi:

- Imeti mora celovito znanje glede razvoja programske opreme.
- Imeti mora znanje in izkušnje o specifikaciji, načrtu ter razvoju programske opreme.
- Sposobna mora biti upravljati z veliko podrobnostmi.
- Razumeti mora tipe napak in kje v kodi se te napake lahko pojavijo.
- Razmišljati mora kot znanstvenik in postaviti hipotezo, ki se navezuje na obstoj določene napake.
- Dobro mora poznati programsko opremo, ki jo testira. To znanje lahko pride iz izobraževanja, predstavitev ali pa iz delovnih izkušenj.
- Ustvariti in dokumentirati mora testne scenarije. Pri kreiranju testnih scenarijev mora izbrati kar se da široko množico vhodnih podatkov. Izbrani naj bi bili podatki, pri katerih je največja verjetnost, da povzročijo napako. Poznavanje aplikacije je pri tem ključno.
- Pripraviti mora testni postopek za izvedbo testa.
- Načrtovati mora izvajanje testov in glede na to zagotoviti primerna sredstva.
- Izvajati mora teste in je odgovorna za beleženje rezultatov.
- Preučuti mora rezultate testov in se odločiti, ali jih je programska oprema opravila ali ne. To zahteva razumevanje in sledenje veliki količini podatkov.
- Navaditi se mora uporabljati testna orodja in skrbeti, da ohranja stik z najsodobnejšo tehnologijo.

- Dobro mora sodelovati s svetovalci, ki so pripravili uporabniške zahteve, načrtovalci in razvijalci. Pogosto pa mora navezati stik s stranko ali končnimi uporabniki.
- Biti mora izobražena in mora trenirati svojo specialnost. Prav tako mora posodabljati svoje znanje, da ostane v stiku z vsemi tehnološkimi spremembami.

3.3 Metode testiranja

Pameten tester, ki želi čas in sredstva uporabiti kar se da učinkovito, ve, da mora pred začetkom testiranja pripraviti ustrezen testni scenarij. Z ustreznim testnim scenarijem mislimo na takšen testni scenarij, ki ima dobre možnosti, da razkrije napake v programski opremi. Sposobnost priprave učinkovitega testnega scenarija je ključna v organizacijah, kjer je prisotno zavedanje pomembnosti visoke kakovosti programske opreme. Dober testni proces ima več dobrih posledic. Če so testni scenariji učinkoviti, je:

- večja verjetnost odkritja napake,
- bolj učinkovita izraba sredstev,
- večja verjetnost ponovne uporabe testnega primera,
- večja povezanost med testiranjem in načrtovanjem projektov ter stroškov,
- večja verjetnost dobave visokokakovostne programske opreme.

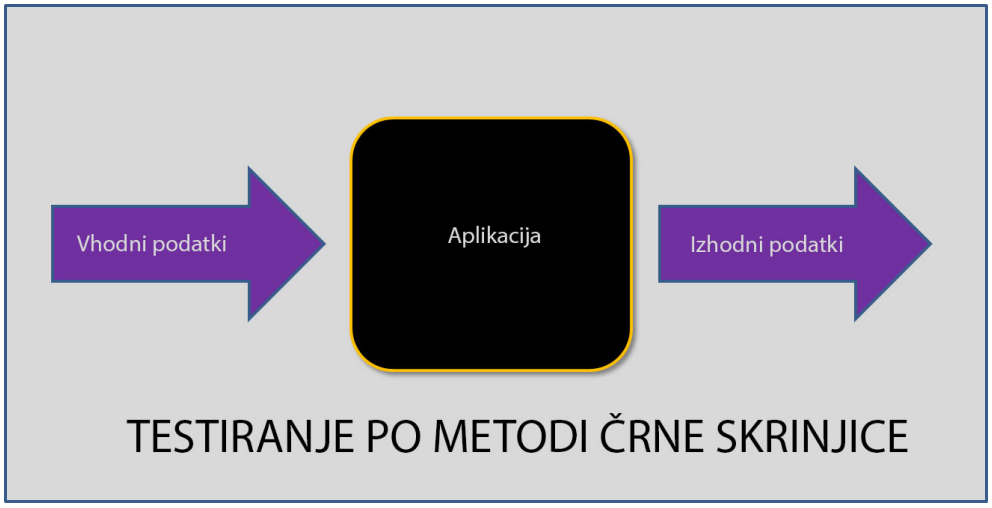
In katero metodo naj bi tester izbral pri načrtovanju učinkovitega testnega scenarija? Da odgovorimo na to vprašanje, moramo pogledati na programsko opremo kot na inženirski produkt. Tu imamo tri tipične metode načrtovanja testnih scenarijev[11].

3.3.1 Metoda črne skrinjice

Pri uporabi metode črne skrinjice oziroma t. i. "black-box test"tester šteje programsko opremo, ki jo testira, kot neprozorno. Nima nobenega znanja o notranji strukturi programske opreme (nima odgovora na vprašanje, kako dela). Edino znanje, ki ga tester ima, je, kaj dela. Velikost programske opreme, ki se testira po tej metodi, je lahko zelo različna. Na primer: od enostavnih modulov do samostojnih aplikacij in celih sistemov. Opis delovanja oziroma funkcionalnosti za programsko opremo, ki se testira, lahko izvira iz uradne specifikacije, diagrama vhodnih in izhodnih podatkov ali pa dobro definiranih pogojev pred izvajanjem in po njem. Drugi vir informacij so uporabniške zahteve, dokument, ki običajno opisuje funkcionalnost programske opreme, ki se testira, vhodne in izhodne podatke. Tester vnese vnaprej določene specifične podatke kot vhodne podatke v programsko opremo, ki se testira, in nato preveri, ali se izhodni podatki ujemajo s tistimi, ki so bili predvideni. Ker ta koncept testiranja preverja predvsem obnašanje programske opreme in njenih

funkcionalnosti, se mu pogosto reče funkcionalni test. Ta koncept je predvsem uporaben za razkrivanje napak v uporabniških zahtevah in funkcionalni specifikaciji[11].

Pri uporabi te metode testiranja moramo na program gledati kot na črno skrinjico. Pri izvajanju te metode se notranjega obnašanja in strukture programa poskušamo ne zavedati. Namesto tega usmerimo pozornost na okoliščine, ki povzročijo, da program deluje v nasprotju s specifikacijami. Cilj testiranja z metodo črne skrinjice je, da se prepričamo, da program rešuje zastavljeni problem.

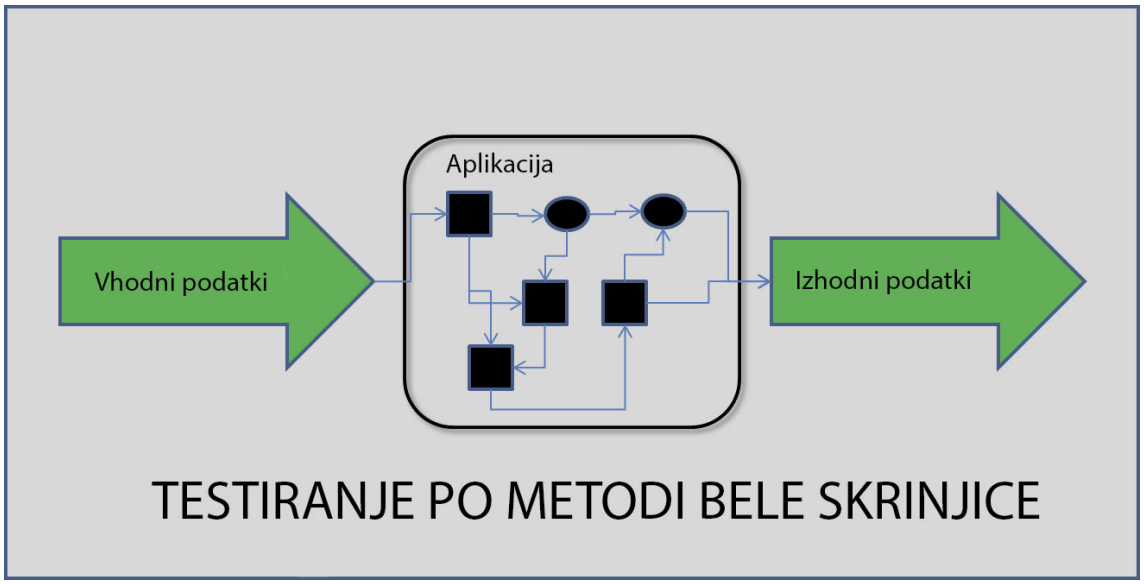


Slika 3.2: Testiranje po metodi črne skrinjice.

3.3.2 Metoda bele skrinjice

Druga metoda testiranja je metoda bele skrinjice oziroma t. i. "white-box test". Pri testiranju po tej metodi je treba pogledati strukturo programa. Testiranje pa se izvaja s podatki, ki bi lahko povzročili napačno delovanje, pridobimo pa jih s pregledom programske kode. Da lahko oseba, ki izvaja test, pripravi testni scenarij po tej metodi, mora imeti znanje o notranji strukturi programske opreme. Imeti mora dostop do izvorne kode oziroma do psevdokode, ki opisuje delovanje. Pri pripravi testnega scenarija oseba, ki testira, nato izbere take primere, ki preverijo pravilnost delovanja posameznih funkcij oz. delov programa. Testi so na primer zasnovani tako, da preverijo vse pogoje ali pa vse veje oz. pogoje pravilno/nepravilno (true/false) znotraj modula ali

funkcije. Glede na to, da je načrtovanje in izvajanje testa po metodi bele skrijice časovno precej zahtevno, je ta koncept praviloma uporabljen pri manjših delih aplikacije, kot so na primer posamezne funkcije ali deli funkcij. Testiranje po konceptu bele skrinjice je predvsem uporabno za odkrivanje napak v kodi, logiki in zaporedju izvajanja ter za napake pri inicializaciji in prenosu podatkov[11].

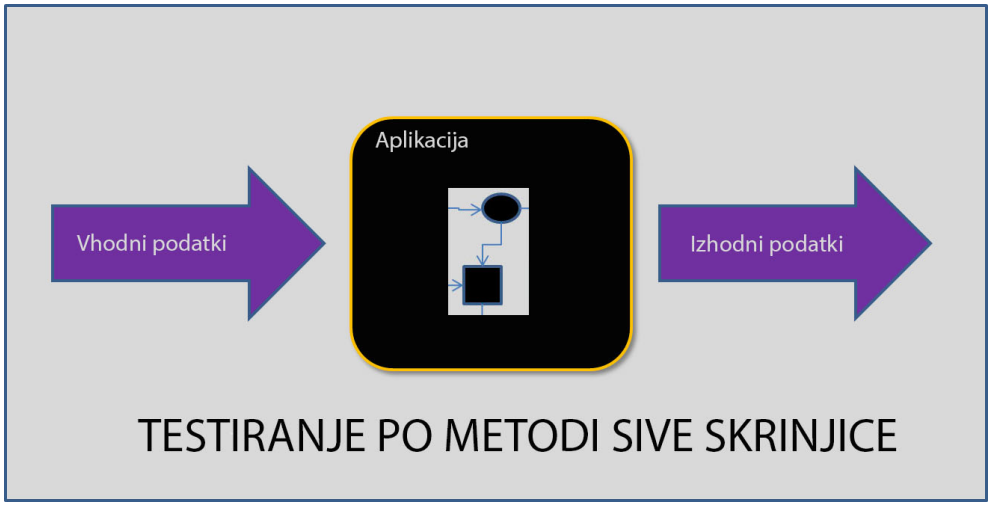


Slika 3.3: Testiranje po metodi bele skrinjice.

Pameten tester ve, da je treba za dosego cilja, to je dobavo programske opreme z malo napakami in visoko kakovostjo, pri pripravi testnega scenarija uporabiti obe zgoraj opisani metodi. Obe metodi silita testerja, da izbere končno mnogo testnih primerov, ki jih je treba preveriti znotraj testnega scenarija. Nobena od metod sama po sebi ne zagotavlja, da bo test razkril vse večje napake pri testiranju. Vendar pa je vsaka od metod dobra za odkrivanje določene vrste napake. Pri testnem scenariju, v katerem sta vsebovani obe metodi testiranja, si tester izboljša verjetnost odkritja napak v programski opremi, ki jo testira. S tem oseba, ki izvaja test, tudi pridobi dober ponovno uporabljiv testni scenarij za primere testiranja popravkov in novih izdaj programske opreme[11].

3.3.3 Metoda sive skrinjice

Če sklepamo iz obeh opisanih metod, vidimo, da imata metodi črne in bele skrinjice vsaka svoje dobre in slabe lastnostni. Zato se v zadnjem času vedno bolj uporablja mešanica obeh metod. To imenujemo metoda sive skrinjice. Tester še vedno izvaja testiranje z vidika končnega uporabnika. Hkrati pa ima dostop do delov notranje zgradbe programske opreme.



Slika 3.4: Testiranje po metodi sive skrinjice.

3.4 Specifika testiranja programske opreme po naročilu

Pri testiranju programske opreme po naročilu ne moremo vedno in v celoti uporabiti že uveljavljenih postopkov, predvsem zato, ker je vsak razvoj programske opreme po naročilu drugačen do predhodnega. Med podobnimi rešitvami lahko potegnemo neke vzporednice oziroma lahko uporabimo predpripravljene vzorce testnih scenarijev. Ko pridemo do podrobnosti, ki so specifične za posamezno rešitev, pa je treba te predpripravljene vzorce ustrezno prilagoditi.

Vzemimo za primer enostavno rešitev, ki na podlagi podatkov pripravi poročilo. Vendar naročnik rešitve A želi na poročilu več podatkov, ki so sortirani po določenih stolpcih. Naročnik podobnega poročila B nad podobno zbirko podatkov pa želi poročilo, na katerem so izpisana samo odstopanja od

vnaprej definiranih norm. Za pripravo testnega scenarija lahko uporabimo predlogo, v kateri so navedene vse naloge, ki so skupne poročilom. Na primer: poročilo se izvede in vrne rezultat, poročilo je mogoče natisniti oziroma izvoziti v podprte izvozne datoteke in tako naprej. Dodatno pa je treba pripraviti specifične naloge za določeno poročilo. Za poročilo A je na primer treba preveriti pravilnost sortiranja, ali sortiranje deluje tudi s posebnimi znaki ... V primeru poročila B pa je treba preveriti, ali so na poročilu dejansko samo zapisi, ki odstopajo od vnaprej definiranih norm.

Izhodišče za pripravo testnega scenarija je zasnova rešitve, na osnovi katere je narejena programska oprema. Zasnova rešitve je dokument, ki natančno opisuje delovanje in funkcionalnosti rešitve. Zasnovo rešitve tipično pripravi član razvojne ekipe, ki ima dovolj široko znanje zadevane tematike, hkrati pa je tudi seznanjen z razvojnimi procesi pri razvoju programske opreme. Za pripravo kakovostne zasnove rešitve je seveda najprej potrebna jasna uporabniška želja, kajti le na podlagi te je mogoče pripraviti zasnovo rešitve in kasneje tudi rešitev, ki bo zadovoljila naročnika. Uporabniške zahteve morajo biti jasno navedene in zabeležene v ločenem dokumentu, ki ga mora naročnik predhodno potrditi. Ta dva dokumenta se tudi ločita po vsebini, ki jo opisujeta. Dokument uporabniških zahtev opisuje predvsem, kaj želi stranka rešiti, če lahko ob tem zajamemo še, zakaj to stranka želi, je to dodaten plus. Dokument zasnove rešitve pa opisuje, kako bo rešitev narejena.

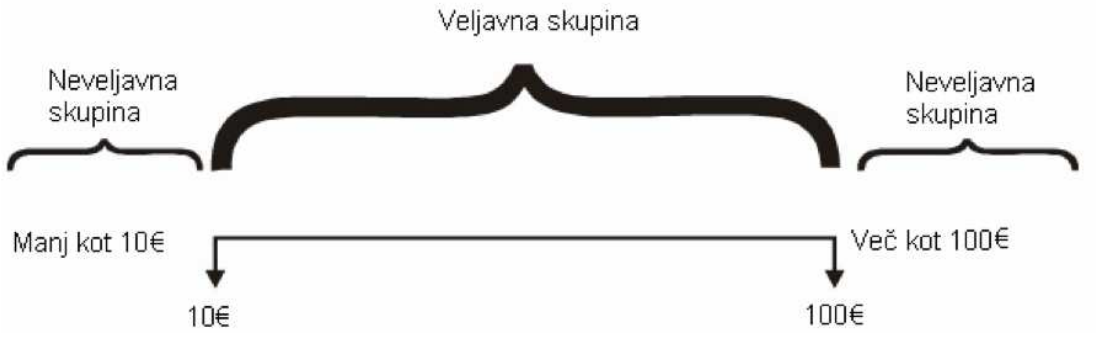
Poglavje 4

Postopki testiranja programske opreme

4.1 Metode črne skrinjice

4.1.1 Določanje ekvivalentnih vhodnih skupin

Pri tej metodi določimo vhodne skupine, ki so enako obravnavane s strani sistema in ustvarijo enak rezultat. Kot primer vzemimo bankomat, ki nam dovoli dvigovati gotovino v bankovcih po 10 EUR, v razponu od 10 EUR do 100 EUR. Imamo 3 ekvivalentne skupine: eno veljavno in dve neveljavni, kar je prikazano na sliki 4.1.



Slika 4.1: Določanje vhodnih skupin.

Če se tester odloči, da bo preizkusil vse možnosti, ker ima malo možnih kombinacij (dvigne 10 EUR, 20 EUR, 30 EUR itd.), bo tako samo tratil svoj čas. Če lahko dvigne 10 EUR, lahko dvigne tudi 20 EUR, 30 EUR ... 100 EUR.

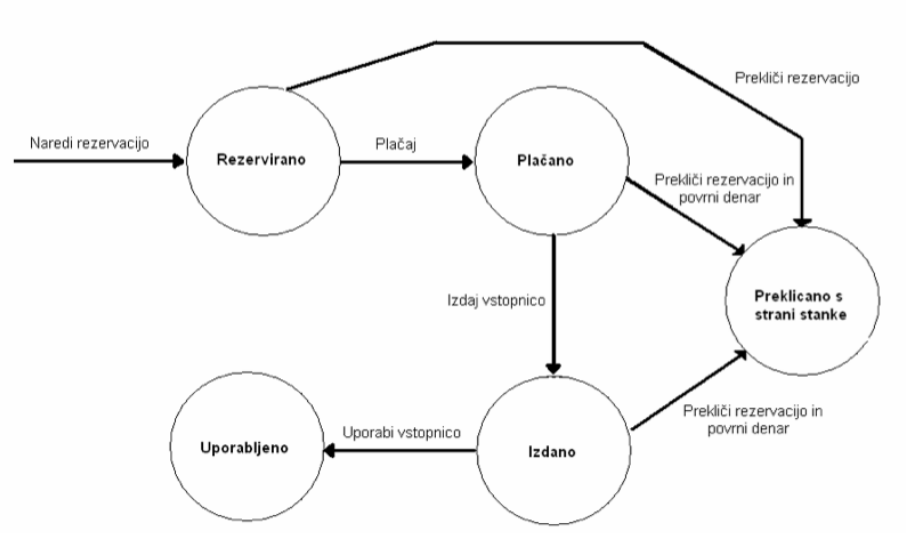
Na enak način lahko preveri tudi pravilnost delovanja znotraj dveh neveljavnih skupin. Če poskusi dvigniti na primer 5 EUR ali 110 EUR, mu jih bankomat ne sme izplačati. Na računu seveda mora biti dovolj denarja, dnevni limit ne sme biti presežen in, ne nazadnje, v bankomatu mora biti dovolj denarja. Seveda predpostavljamo, da štetje denarja deluje brez težav. V tem primeru potrebujemo samo dve neveljavni in eno veljavno vrednost, da je obseg delovanja pokrit[9].

4.1.2 Analiza mejne vrednosti

Izkušnje kažejo, da je metoda analize mejne vrednosti zelo pomembna, saj omejitve velikokrat niso veljavne. Filozofija tega testiranja je preprosta. Če lahko varno hodimo tik ob robu prepada, potem lahko skoraj z gotovostjo trdimo, da bomo varno hodili tudi po sredini polja. V primeru bankomata bi izbrali naslednje vrednosti: 0 EUR – vrednost izpod spodnje meje, 10 EUR – spodnja meja, 100 EUR – zgornja meja in 110 EUR – vrednost nad zgornjo mejo[9].

4.1.3 Diagrami prehodov stanj

Diagram prehodov stanj je zelo stara metoda, ki pa je še vedno učinkovita pri opisovanju modela sistema, in je vodnik pri testiranju. Diagram opisuje sistem (ali komponento), čigar funkcionalnost in rezultat nista izključno odvisna od trenutnega vhoda, temveč tudi od prejšnjih vhodov. Rezultat prejšnjega vhoda se imenuje stanje in prehodi so naloge, ki povzročijo premike iz enega v drugo stanje. Slika 4.2 prikazuje diagram prehodov stanj na primeru rezervacije vstopnic za kino[9].



Slika 4.2: Diagram prehodov stanj.

4.1.4 Metoda naključnega iskanja napak

Velikokrat so nekateri ljudje že po naravi bolj vešči pri testiranju programske opreme. Brez posebnih metod jim uspe izslediti napake v rešitvi. Ena izmed razlag tega pojava je, da ti ljudje podzavestno vadijo tehniko, ki se imenuje naključno iskanje napak. Ko se lotijo testiranja, sledijo svoji intuiciji in izkušnjam. Tako ustvarijo testni scenarij, ki je uspešen pri izsleditvi določenih vrst napak. Za to metodo je zelo težko določiti postopek izvajanja, saj je v veliki meri odvisen od intuicije in testiranja ad-hoc. Osnovna ideja je, da sestavimo listo možnih napak ali situacij, v katerih se bodo napake pokazale, in zanje oblikujemo testne scenarije[9].

4.2 Metode bele skrinjice

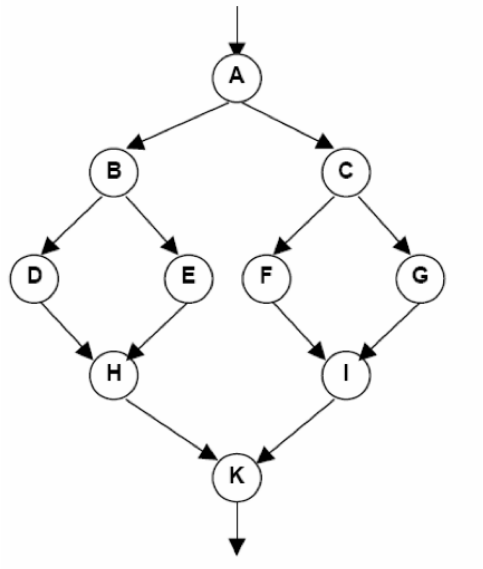
Metode bele skrinjice nam dovoljujejo raziskati notranjo strukturo programa. Pri teh metodah se testne informacije pridobijo iz pregleda logike programa. Brez pogleda v skrinjico ni možno preveriti določenih načinov delovanja rešitve. Raje, kot da poskušamo testirati vsako možno pot, uporabimo različne metode bele skrinjice, pri katerih so na voljo mehanizmi za izbiro posameznih poti testa. Pogosto so metode bele skrinjice povezane z merami za pokritost testiranja, ki merijo odstotek izbranih poti, izvršenih v določenem testnem

scenariju. Projekti imajo po navadi določene stopnje pokritosti testiranja, ki so razvijalcem in testerjem vodilo pri tem, kako temeljito je treba testirati rešitev[9].

Pokritost stavkov, vej in poti so metode bele skrinjice, ki izkoristijo logični tok programa za izdelavo testnih scenarijev. Logični tok je način, kako se lahko deli programa izvajajo, ko ga poženemo. Logični tok programa lahko predstavimo z grafom toka izvajanja programa, kar vidimo na sliki 4.3.

Graf sestavljajo:

- vozlišča – predstavljajo stavke, ki so lahko uporabljeni pri izvajanju programa;
- povezave – predstavljajo pot, po kateri logika omogoča programu prehod od enega stavka k drugemu;
- odločitvena vozlišča – to so vozlišča, ki imajo več kot en izhod;
- odločitvene povezave – to so povezave, ki zapuščajo odločitveno vozlišče;
- poti – možne poti, po katerih se lahko premikamo od enega vozlišča do drugega po povezavah.

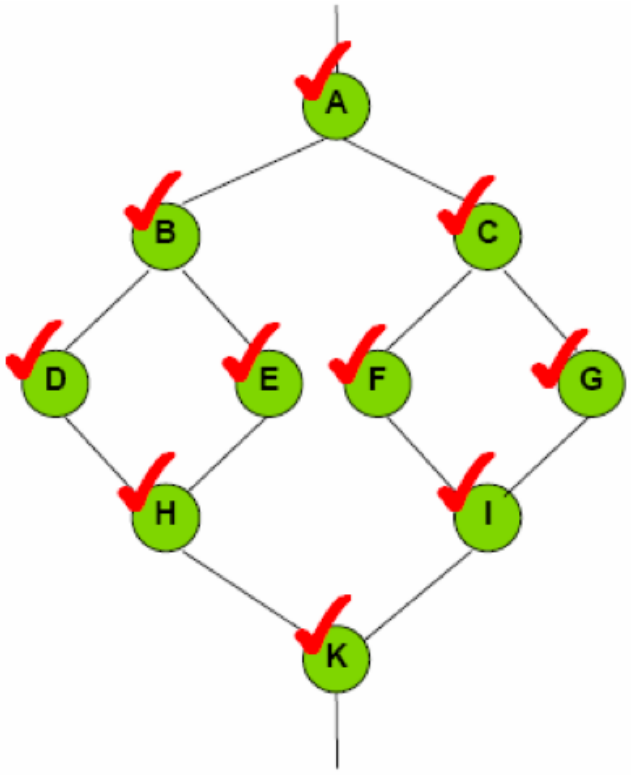


Slika 4.3: Graf toka izvajanja programa.

Izvajanje vsakega testnega scenarija povzroči, da program izvede določen ukaz, ki se odraža v izbiri specifične poti v grafu toka izvajanja programa. Različne poti lahko izbiramo s spreminjanjem pogojev, ki povzročijo drugačno izbiro povezav vozlišč, torej izberemo drugačno pot[14].

4.2.1 Pokritost stavkov

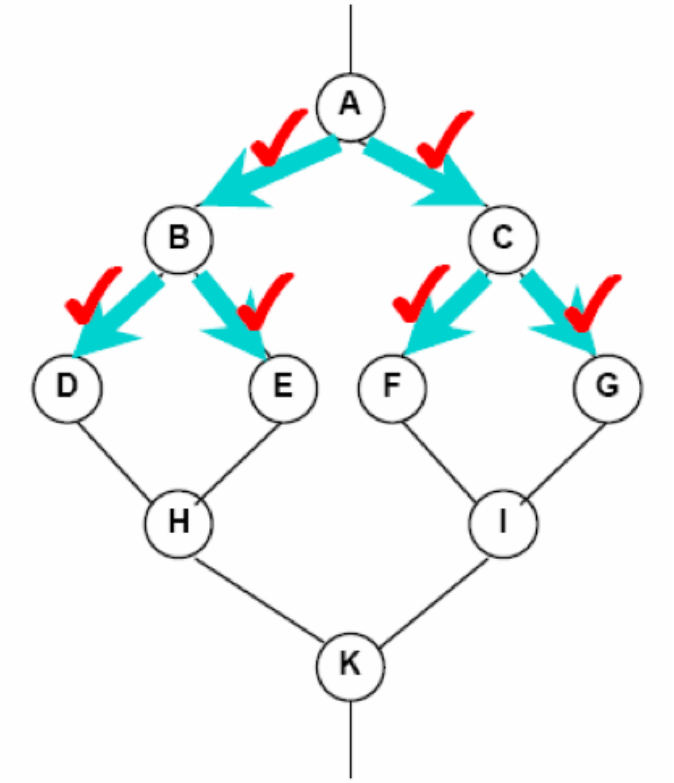
Pokritost stavkov je določena z oceno deleža stavkov, ki so pokriti z določenim testom. 100-odstotno pokritost stavkov dosežemo tako, da je vsak stavek v programu vsebovan vsaj v enem testnem scenariju. Pokritost stavkov se ujema z obiskom vozlišč na grafu. V grafu (slika 4.4) imamo 10 vozlišč. Če izberemo pot A, B, D, H, K, pokrijemo 5 od 10 vozlišč, kar pomeni 50-odstotno pokritost[14].



Slika 4.4: Pokritost vozlišč.

4.2.2 Pokritost povezav

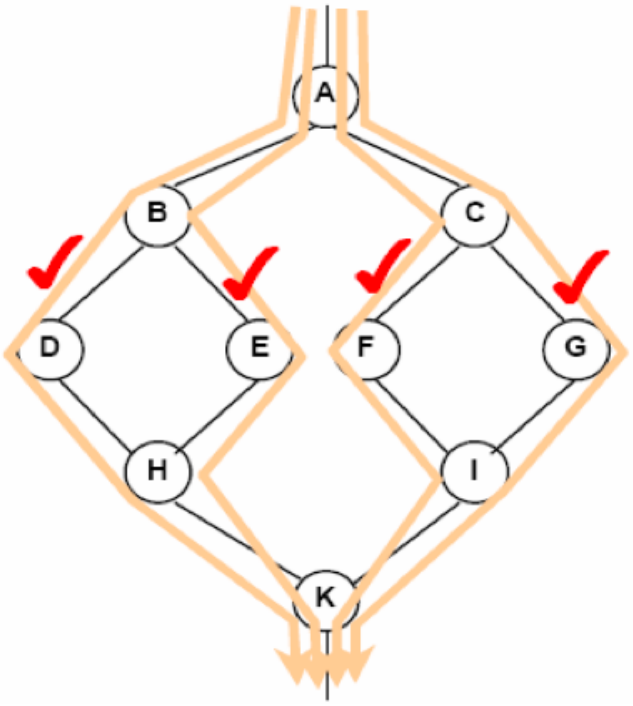
Pokritost povezav je določena z oceno deleža odločitvenih povezav, ki so pokrite v testu. 100-odstotno pokritost dosežemo tako, da je vsaka odločitvena povezava vsebovana v vsaj enem testnem scenariju. Pokritost povezav se ujema z obiskom odločitvenih vozlišč v grafu. 100-odstotna pokritost je tam, kjer so vse odločitvene povezave obiskane v testnem scenariju. Slika 4.5 prikazuje 6 odločitvenih povezav, če izberemo pot A, B, D, H, K, pokrijemo 2 od 6 robov vej, kar pomeni 33-odstotno pokritost[14].



Slika 4.5: Pokritost povezav.

4.2.3 Pokritost poti

Pokritost poti je določena z oceno deleža poti, ki so pokrite z določenim testom. 100-odstotno pokritost poti dosežemo tako, da je vsaka pot v programu vsebovana vsaj v enem testnem scenariju. Pokritost poti se ujema z obiskom poti v grafu. 100-odstotna pokritost je tam, kjer so vse poti obiskane v testnem scenariju. Na grafu, ki ga prikazuje slika 4.6, so 4 poti. Če izberemo pot A, B, D, H, K, pokrijemo 1 od 4 poti, kar pomeni 25-odstotno pokritost[14].



Slika 4.6: Pokritost poti.

4.3 Primerjava metode črne in bele skrinjice

Medtem ko sta obe metodi uspešni pri zagotavljanju pravilnega delovanja sistema, lahko samo metoda bele skrinjice potrdi, da je bil proces izvedbe pravilen. Običajno menimo, da je ob pravilnem rezultatu testa pravilen tudi proces obdelav, kar pa seveda ni vedno res. V nekaterih primerih je možno, da dobimo pravilen rezultat testa z napačnim procesom. Ta fenomen je poznan kot naključna pravilnost, ki ga z uporabo strategije črne skrinjice ne odkrijemo vedno. Dober testni scenarij tako vključuje metode iskanja napak po metodi bele in črne skrinjice[9].

4.4 Izvedba testiranja

Testiranje lahko izvedemo ročno ali s pomočjo za to namenjenih orodij.

4.4.1 Ročno testiranje

Ročno testiranje sreča na začetku poti vsak preizkuševalec, pa tudi kasneje, ko je že izkušen, ne gre brez njega. Uporabimo ga povsod tam kjer[7]:

- ni dovolj časa za avtomatizacijo,
- ni izkušenega kadra, ki bi obvladal postopke avtomatizacije,
- avtomatizacija ni smiselna zaradi prevelikih stroškov in neponovljivosti,
- ko testiramo uporabniški vmesnik ali
- določamo izvor in ponovitev napake.

4.4.2 Avtomatsko testiranje

V praksi se pogosto dogaja, da moramo določene teste večkrat ponavljati. Kadar odkrijemo kakšno napako, je običajno, ko je odpravljena, treba določen del programske opreme testirati večkrat, da lahko potrdimo, da je napaka zares odpravljena in se niso pojavile nove napake. V tem primeru moramo presoditi, ali bomo take teste opravljali ročno ali pa jih bomo avtomatizirali in si pomagali s testnimi orodji. Prednosti avtomatizacije testiranja so[8]:

- hitrost - sodobna orodja omogočajo obdelavo ogromne količine podatkov v zelo kratkem času,

- učinkovitost - če testiramo ročno, v tem času ne moremo delati nič drugega; z avtomatizacijo prihranimo čas, ki ga lahko porabimo npr. za planiranje testiranja,

- točnost in natančnost - ko izvedemo na stotine testov, človekova koncentracija pade, pojavljajo se napake, medtem ko orodje izvaja isti test in preverja rezultate z vedno isto natančnostjo in točnostjo,

- vztrajnost - testno orodje se nikoli ne utrudi, teste lahko opravlja vedno znova.

Orodij za izvajanje avtomatskega testa je na trgu zelo veliko. Naštejmo samo nekatere od njih:

- Test Studio:
  Rešitev podjetja Telerik je namenjena testiranju namiznih in spletnih aplikacij. Deluje samo na operacijskem sistemu Microsoft Windows.

- TestComplete:
  Tako kot Test Studio je tudi rešitev TestComplete namenjena testiranju namiznih in spletnih aplikacij. Podpira pa tudi testiranje mobilnih aplikacij razvitih s spletnimi tehnologijami.

- Selenium:
  Gre za odprtokodno rešitev, ki je brezplačna za uporabo. Deluje na na vseh operacijskih sistemih, ker je zasnovana kot vtičnik za spletni brskalnik. Namenjena je testiranju spletnih aplikacij.

- HP QuickTest Professional:
  Rešitev podjetja HP je namenjena zgolj testiranju namiznih aplikacij v Microsoft Windows okolju.

- Testing Anywhere:
  Zanimivost te rešitve je, da poleg namiznih in spletnih aplikacij podpira tudi mobilne. Trenutno podprti platformi sta Android in iOS.

Pri izbiri orodja moramo upoštevati več faktorjev, kot so na primer cena orodja in vzdrževanja, zahtevnost priprave in vzdrževanja testnih scenarijev in kakovost poročila testa.

4.5 Posebnosti pri rešitvah po naročilu

Glede na to, da so rešitve po naročilu tipično projekti, ki se razvijajo samo enkrat, uporaba avtomatskega testiranja ne prinese bistvene prednosti. Čas, ki bi ga porabili za pripravo testnega scenarija znotraj orodja za testiranje, se ne razlikuje zelo od časa, ki ga porabimo za ročno testiranje. Tako pri testiranju rešitev po naročilu še vedno uporabljamo ročno testiranje. To pa mora biti seveda izvedeno na podlagi testnega scenarija. Pri pripravi teh se v veliki meri naslanjamo na dokumente uporabniških zahtev, zasnove rešitve in na izkušnje osebe, ki pripravlja testne scenarije.

Poglavje 5

Praktični primer zagotavljanja kakovosti

V tem sklopu si bomo pogledali primer zagotavljanja kakovosti programske rešitve po na naročilu. Pred tem pa si moramo pogledati, kaj sploh rešitve po naročilu so in kako jih rešujemo. Rešitev po naročilu se vedno začne na pobudo stranke, ki ima določene zahteve, ki se s standardnim produktom ne morejo pokriti. Vsaka rešitev po naročilu gre skozi svoj življenjski cikel razvoja programske opreme. Začne se z željo stranke, za katero se potem pripravita analiza zahtev in specifikacija želja. V tej fazi nastane dokument uporabniške zahteve. Po potrditvi uporabniških zahtev s strani stranke se začne načrtovanje rešitve, katerega rezultat je dokument zasnove rešitve. Na osnovi tega dokumenta lahko stranka naroči rešitev. Sama rešitev je lahko na primer karkoli - od samostojne aplikacije, servisne aplikacije, spletne aplikacije, mobilne aplikacije ali pa SQL-poizvedbe. Po naročilu se začne z implementacijo in testiranjem rešitve. Ko je rešitev v celoti pripravljena, se preda stranki v uporabo.

5.1 Uporabniške zahteve

Želja stranke je izvoz obračunskih podatkov v podatkovno bazo. Na podlagi tega izvoza želijo pripravljati poročila. Podatki se prenesejo v tabelo na zahtevo. Poročilo naj vsebuje podatke za vse osebe, ki so zabeležene v sistemu. Podatki, ki jih potrebujejo, so naslednji:

- datum podatkov,
- matična (evidenčna) številka zaposlenega,
- ime zaposlenega,
- priimek zaposlenega,
- organizacijska enota,
- oddelek,
- pododdelek,
- skupna prisotnost – v obliki hh:mm,
- skupna prisotnost – v minutah,
- plan – v obliki hh:mm,
- plan – v minutah,
- nadurno delo – v obliki hh:mm,
- nadurno delo – v minutah,
- izobraževanja – v obliki hh:mm,
- izobraževanja – v minutah,
- odsotnost – bolniška – v obliki hh:mm,
- odsotnost – bolniška – v minutah,
- odsotnost - dopust – v obliki hh:mm,
- odsotnost - dopust – v minutah.

5.2 Zasnova rešitve

Opis rešitve

Modul "Izvoz obračuna" je samostojni modul, ki deluje znotraj sistema za registracijo delovnega časa. Njegova naloga je obračun podatkov za zaposlene in zapis tega v posebno tabelo v podatkovno bazo.

Delovanje rešitve

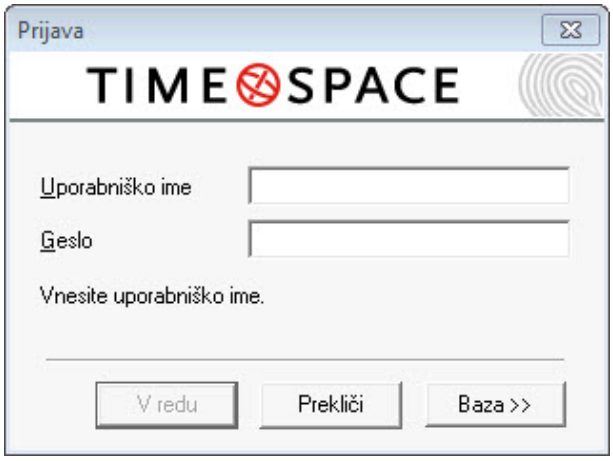
Modul je sestavljen kot konzolna aplikacija z grafičnim vmesnikom. Dodani opcijski parametri ob izvedbi iz ukazne vrstice določajo obdobje, ki naj ga modul obdela. Parametri, ki so na voljo, so včeraj, tekoči mesec, prejšnji mesec in tekoče leto. Zagon brez parametrov odpre grafični vmesnik, v katerem so nastavitve izvoza. V grafičnem vmesniku je omogočeno izvajanje izvoza za poljubno obdobje. Podatki se izvozijo v tabelo PP_CALCULATION_EXPORT v podatkovni bazi sistema. Dan je definiran na enak način kot v sistemu, kar pomeni, da zajema tudi delo čez polnoč. Ob večkratnem obračunu za isti dan se predhodni obračunski podatki izbrišejo. Modul obdela vse zaposlene, vnesene v sistem, ki so v obdobju obdelave vsaj en dan aktivni. Za zaposlenega se za vsak dan znotraj obdobja izvajanja doda po en zapis v izvozno tabelo. Zapis vsebuje naslednje podatke:

- datum,
- referenčna številka zaposlenega,
- ime zaposlenega,
- priimek zaposlenega,
- ime organizacijske enote,
- oddelek,
- pododdelek,
- skupna prisotnost – v obliki hh:mm in v minutah; skupna prisotnost zaposlenega iz sistema,
- plan – v obliki hh:mm in v minutah; plan zaposlenega iz sistema,

- nadurno delo – v obliki hh:mm in v minutah; dnevni saldo zaposlenega iz sistema,
- izobraževanja – v obliki hh:mm in v minutah; seštevek nastavljenih kategorij iz sistema,
- odsotnost – bolniška – v obliki hh:mm in v minutah; seštevek nastavljenih kategorij iz sistema,
- odsotnost - dopust – v obliki hh:mm in v minutah; seštevek nastavljenih kategorij iz sistema.

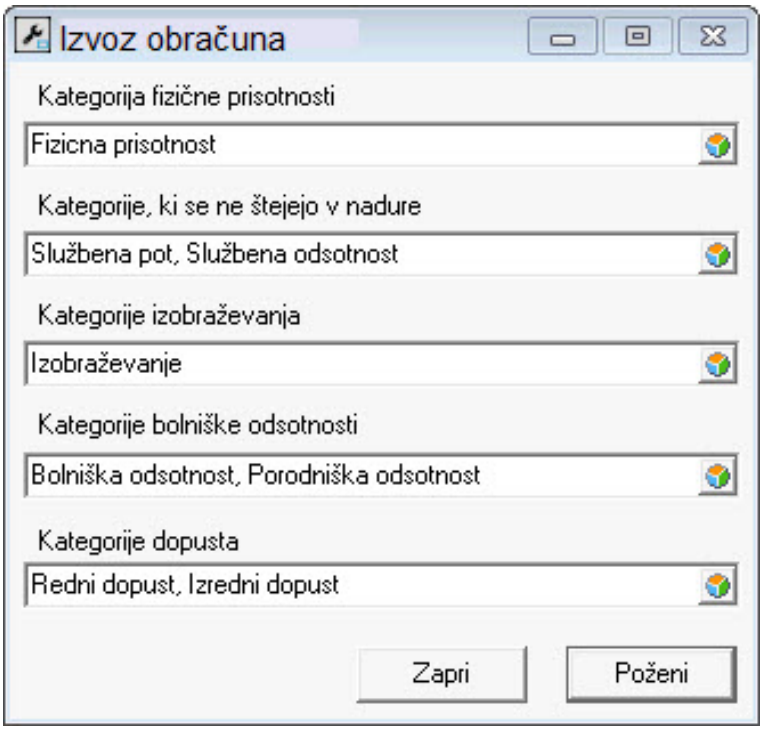
5.3 Rešitev

Rešitev je samostojna aplikacija, ki izvozi obračunske podatke v podatkovno bazo. Izvršitev aplikacije brez dodatnih parametrov odpre prijavno okno.



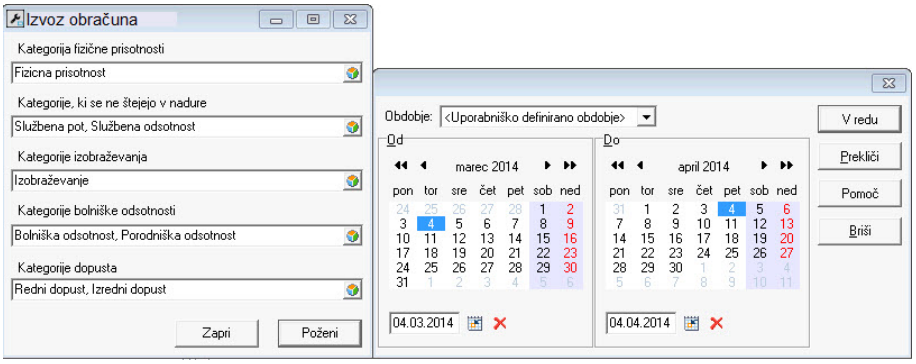
Slika 5.1: Prijavno okno.

Po uspešni prijavi v sistem se pokaže okno z nastavitvami, ki jih uporabnik lahko izbira.



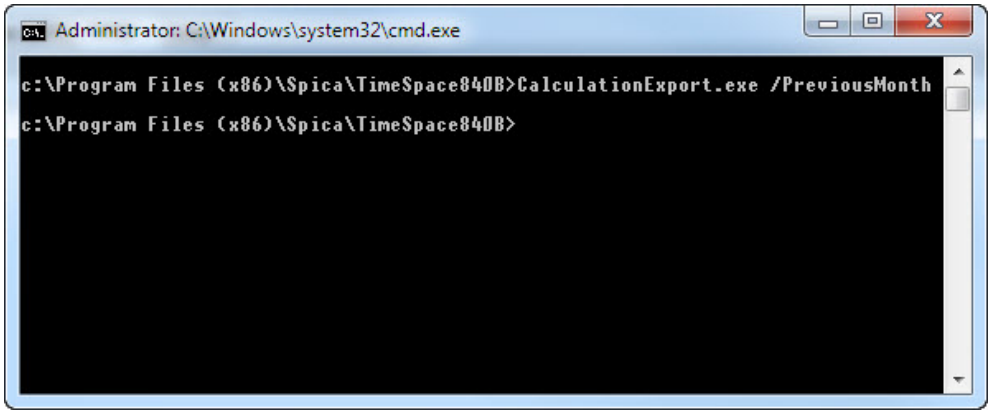
Slika 5.2: Okno z nastavitvami aplikacije.

V grafičnem načinu lahko uporabnik izvede izvoz podatkov za poljubno obdobje. V tem primeru se mu pokaže okno za izbiro obdobja.



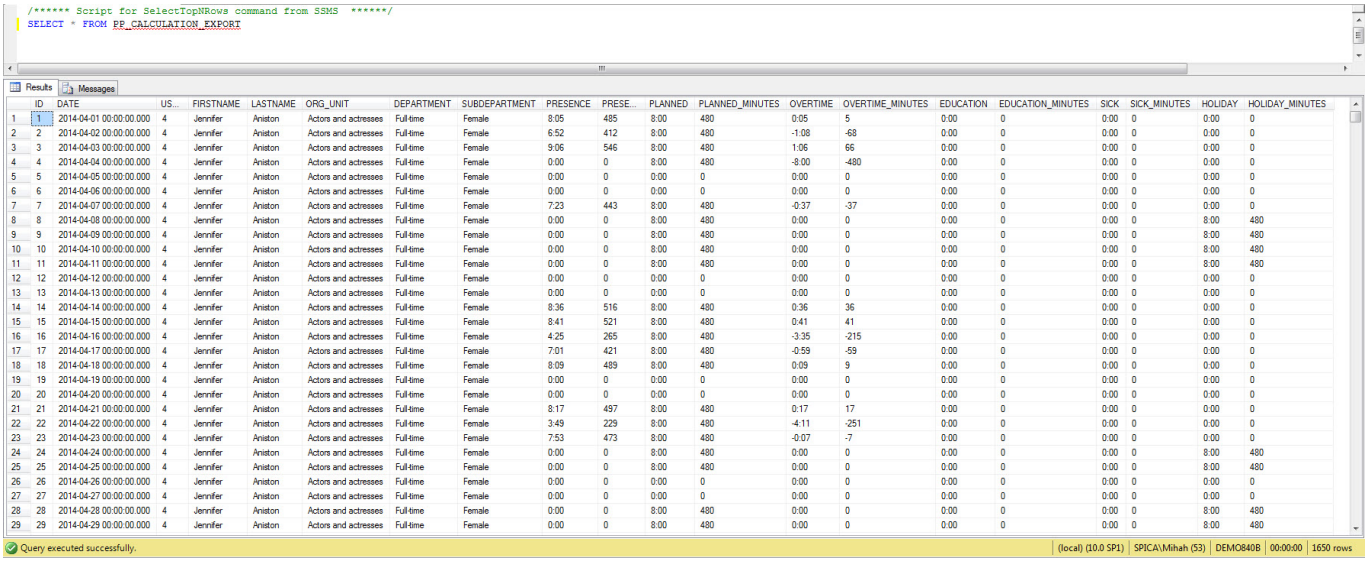
Slika 5.3: Okno za izbiro obdobja.

Aplikacijo je mogoče izvršiti tudi preko ukazne vrstice. To naredimo tako, da dodamo imenu aplikacije parameter, ki opisuje časovno izvajanje aplikacije. Izvajanje preko ukazne vrstice je namenjeno avtomatiziranemu izvajanju aplikacije preko razporejevalnika opravil.



Slika 5.4: Primer izvajanja preko ukazne vrstice.

Rezultat izvajanja aplikacije so podatki v tabeli v podatkovni bazi.



```
/****** Script for SelectTopNRows command from SSMS  ******/
SELECT * FROM PP_CALCULATION_EXPORT
```

Results | Messages

	ID	DATE	US...	FIRSTNAME	LASTNAME	ORG_UNIT	DEPARTMENT	SUBDEPARTMENT	PRESENCE	PRESE...	PLANNED	PLANNED_MINUTES	OVERTIME	OVERTIME_MINUTES	EDUCATION	EDUCATION_MINUTES	SICK	SICK_MINUTES	HOLIDAY	HOLIDAY_MINUTES
1	1	2014-04-01 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	8:05	485	8:00	480	0:05	5	0:00	0	0:00	0	0:00	0
2	2	2014-04-02 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	6:52	412	8:00	480	-1:08	-68	0:00	0	0:00	0	0:00	0
3	3	2014-04-03 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	9:06	546	8:00	480	1:06	66	0:00	0	0:00	0	0:00	0
4	4	2014-04-04 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	8:00	480	-8:00	-480	0:00	0	0:00	0	0:00	0
5	5	2014-04-05 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0
6	6	2014-04-06 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0
7	7	2014-04-07 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	7:23	443	8:00	480	-0:37	-37	0:00	0	0:00	0	0:00	0
8	8	2014-04-08 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	8:00	480	0:00	0	0:00	0	0:00	0	8:00	480
9	9	2014-04-09 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	8:00	480	0:00	0	0:00	0	0:00	0	8:00	480
10	10	2014-04-10 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	8:00	480	0:00	0	0:00	0	0:00	0	8:00	480
11	11	2014-04-11 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	8:00	480	0:00	0	0:00	0	0:00	0	8:00	480
12	12	2014-04-12 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0
13	13	2014-04-13 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0
14	14	2014-04-14 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	8:36	516	8:00	480	0:36	36	0:00	0	0:00	0	0:00	0
15	15	2014-04-15 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	8:41	521	8:00	480	0:41	41	0:00	0	0:00	0	0:00	0
16	16	2014-04-16 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	4:25	265	8:00	480	-3:35	-215	0:00	0	0:00	0	0:00	0
17	17	2014-04-17 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	7:01	421	8:00	480	-0:59	-59	0:00	0	0:00	0	0:00	0
18	18	2014-04-18 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	8:09	489	8:00	480	0:09	9	0:00	0	0:00	0	0:00	0
19	19	2014-04-19 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0
20	20	2014-04-20 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0
21	21	2014-04-21 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	8:17	497	8:00	480	0:17	17	0:00	0	0:00	0	0:00	0
22	22	2014-04-22 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	3:49	229	8:00	480	-4:11	-251	0:00	0	0:00	0	0:00	0
23	23	2014-04-23 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	7:53	473	8:00	480	-0:07	-7	0:00	0	0:00	0	0:00	0
24	24	2014-04-24 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	8:00	480	0:00	0	0:00	0	0:00	0	8:00	480
25	25	2014-04-25 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	8:00	480	0:00	0	0:00	0	0:00	0	8:00	480
26	26	2014-04-26 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0
27	27	2014-04-27 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0	0:00	0
28	28	2014-04-28 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	8:00	480	0:00	0	0:00	0	0:00	0	8:00	480
29	29	2014-04-29 00:00:00.000	4	Jennifer	Aniston	Actors and actresses	Full-time	Female	0:00	0	8:00	480	0:00	0	0:00	0	0:00	0	8:00	480

Query executed successfully. | (local) (10.0 SP1) | SPICA\Mihah (53) | DEMO840B | 00:00:00 | 1650 rows

Slika 5.5: Rezultat izvajanja aplikacije.

5.4 Izpolnjen testni scenarij

Testni scenarij - izvoz podatkov

Splošne informacije:

- Opercijski sistem: Windows 7 64-bit
- Verzija podatkovnega strežnika: MS SQL 2008 R2
- Uporabljene podatkovne baze: DEMO
- Tester: Miha Herblan
- Datum izvajanja testa: 4. 4. 2014

Dokumentacija:

- Preveri skladnost zahtev, opisanih v dokumentih uporabniških zahtev, in zasnove rešitve z uporabniških navodil:
  Uporabniška navodila se skladajo z željami v uporabniških navodilih in s specifikacijo iz zasnove rešitve.
- Preveri uporabnost uporabniških navodil:
  Uporabniška navodila so jasna in pokrivajo vse dele rešitve.

Glavne funkcionalnosti:

- Rešitev se izvede brez sistemskih napak:
  Rešitev se izvede brez težav.
- Rešitev med delovanjem ne porablja preveč sistemskih sredstev in jih po zaprtju sprosti:
  Rešitev deluje po pričakovanjih. Poraba delovnega spomina ni presegla 500 MB. Obremenitev procesorja ni presegla 20 odstotkov. Testirano s 50, 100 in 500 zaposlenimi.
- Rešitev ima žurnalsko datoteko:
  V mapi, kjer je nameščena rešitev, se ustvari žurnalska datoteka. Podatkov v žurnalski datoteki je dovolj za pregled delovanja rešitve.
- Avtentikacija in omejitve različnih uporabnikov:
  Avtentikacija uporabnikov deluje. Prav tako delujejo omejitve uporabnikov. Preverjeno je delovanje na vseh tipih uporabnikov in z različnimi omejitvami (po enoti, oddelku, pododdelku in organizacijski strukturi).

- Rešitev je v skladu s standardi celostne podobe sistema:
  Rešitev je izdelana v skladu s standardi celostne podobe sistema.

Posebnosti:

- Podatki se izvozijo za pravilno skupino zaposlenih:
  Podatki so pripravljeni za pravo skupino zaposlenih glede na omejitve uporabnika. Testirano z vsemi različnimi skupinami zaposlenih.

- Izvoženi podatki so pravilni:
  Podatki, ki so zapisni v tabeli, so pravilni in odražajo pravilne vrednosti iz sistema. Ročno sem preveril pravilnost vseh izvoženih podatkov za deset zaposlenih, ki so imeli vnesene dodatne dogodke. Preveril sem tudi deset naključno izbranih zaposlenih.

- Podatki se pravilno izvozijo pri delu čez polnoč:
  Podatki pri obračunu vrednosti dela čez polnoč so pravilni. Preveril sem obračun različnih kategorij pri delu čez polnoč.

- Pri večkratnem izvajanju za isto obdobje se predhodni podatki prepišejo:
  Prepisovanje dogodkov deluje. Preveril sem popolno in delno prekrivanje obdobja obdelave.

- Izvajanje preko ukazne vrstice deluje:
  Izvajanje rešitve preko ukazne vrstice deluje brez težav. Preveril sem vse različne parametre. Prav tako sem preveril izvajanje preko sistemskega razporejevalnika opravil.

Ostalo:

Pri testiranju za daljše obdobje (60 dni) in večje število zaposlenih (500 zaposlenih) sem opazil, da izvajanje traja več kot dvajset minut. Stranko je na to treba opozoriti.

Velikost žurnalske datoteke narašča relativno hitro.

Poglavje 6

Zaključek

Vloge kakovostne programske opreme in učinkovitih storitev se vse bolj zavedajo podjetja, ki želijo biti uspešna na trgu. Podjetja, ki se ukvarjajo z razvojem programskih rešitev, morajo zagotoviti, da njihova programska oprema deluje čim bolj brezhibno in izpolnjuje zahteve, ki so bile postavljene pred začetkom razvoja.

S povečevanjem zahtevnosti in velikosti programske opreme je izdelava te brez napak oziroma z malo napakami praktično neizvedljiva, tudi za najboljše razvijalce. Zato je treba zagotoviti v vsakem razvojnem ciklu zadosten obseg testiranja, kar pa ni vedno samoumevno.

Tako kot pri vsakem projektu se tudi pri razvoju programske opreme soočamo s tremi ključnimi komponentami, ki pa se med seboj izključujejo: s časom, z denarjem in s kakovostjo. Praktično se nikoli ne zgodi, da bi imeli na voljo vse te tri komponente. Glede na to, da se testiranje tipično izvaja na koncu razvojnega cikla programske opreme, je največkrat na udaru zaradi bližajočega se dobavnega roka in preseženih načrtovanih stroškov. Zato potrebujemo dober testni scenarij, da lahko sledimo delu in ga učinkovito opravimo. Uporabljamo tudi izvajanje testa že zaključenih smiselnih celot med samim razvojem.

Z vse večjo uporabo programske opreme pri kritičnih operacijah (na primer: medicina, telekomunikacije, finance) in tudi med vsakdanjimi opravili (na primer: kontrola pristopa, registracija delovnega časa), se potreba po visoki kakovosti programske opreme samo še povečuje. In ker vsaka še tako majhna napaka lahko povzroči ogromno škodo, se testiranja programske opreme ne moremo več lotevati z naključnim lovljenjem napak. Za to je potreben načrten in discipliniran pristop pri preprečevanju, iskanju in prijavljanju napak.

Priprava testnega scenarija, tudi za majhen projekt, predstavlja obsežno nalogo, ki je ne smemo podceniti. Planiranje testiranja je naloga, ki naj bi

vključevala celotno testno ekipo in vse ključne osebe iz razvojne ekipe.

Posebno specifiko pri programski opremi predstavljajo rešitve po naročilu. Ker gre tipično za projekte, ki se naredijo enkrat, je treba v določeni meri reciklirati testne scenarije. Pri pripravi testnih scenarijev se v veliki meri naslanjamo na dokument uporabniških zahtev in zasnove rešitve. Seveda so pri pripravi testnih scenarijev zelo pomembne tudi izkušnje testerja, njegovo poznavanje celotnega sistema, znotraj katerega bo rešitev po naročilu delovala in poznavanje želja naročnika. Le če so vsi pogoji izpolnjeni, lahko naročniku dobavimo rešitev, ki bo delovala po pričakovanjih in je brez napak.

Slike

Literatura

[1] (2014) List of software bugs Dostopno na:
http://en.wikipedia.org/wiki/List_of_software_bugs

[2] (2014) Software Errors Cost U.S. Economy $59.5 Billion Annually. Dostopno na:
http://www.ashireporter.org/articles/articles.aspx?id=740

[3] (2014) Opredelitev življenjskega cikla programske opreme. Dostopno na:
http://colos1.fri.uni-lj.si/ERI/RACUNALNISTVO/INFORMATIKA/opredelitev_ivljenjskega_cikla_programske_opreme.html

[4] (2014) Faze življenjskega cikla programske opreme. Dostopno na:
http://colos1.fri.uni-lj.si/ERI/RACUNALNISTVO/INFORMATIKA/faze_ivljenjskega_cikla_programske_opreme.html

[5] (2014) Quality Assurance and Quality Control. Dostopno na:
http://asq.org/learn-about-quality/quality-assurance-quality-control/overview/overview.html

[6] ISO/IEC 25010:2011, *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.*

[7] (2014) Testiranje informacijskega sistema v različnih razvojnih fazah. Dostopno na:
http://www.cek.ef.uni-lj.si/magister/goltez2767.pdf

[8] (2014) Avoiding Shelfware: A Managers' View of Automated GUI Testing. Dostopno na:
http://kaner.com/pdfs/shelfwar.pdf

[9] (2014) Testiranje programske opreme in izdelava načrta za testiranje Time&Space sistema. Dostopno na: http://www.cek.ef.uni-lj.si/u_diplome/kocevar2537.pdf

[10] G. J. Myers, *The art of software testing*, New Jersy: John Wiley & Sons, 2004, str. 8-123.

[11] I. Burnstein, *Practical software testing: a process-oriented approach*, New York: Springer-Verlag, 2003, str. 20-187.

[12] (2014) IEEE Standard Glossary of Software Engineering Terminology. Dostopno na: http://www.idi.ntnu.no/grupper/su/publ/ese/ieee-se-glossary-610.12-1990.pdf

[13] Craig D. R., Jaskiel S. P.: *Systematic Software Testing*, Boston: Artech House, 2002. 58 str.

[14] K. Ross, *Practical guide to software system testing*, West Burleigh: K. J. Ross & Associates, 1998. 9-96 str.

[15] Spica International d.o.o., *Interna dokumentacija Spica Internationa d.o.o.*, 2014.